

**Operating System and Design (19CS2106S)****Lab- 10****Pre-Lab**

**shmat()**: Attach the process to the already created shared memory segment **shmdt()** : Detach the process from the already attached shared memory segment **shmctl()**: Control operations on the shared memory segment

1. **pthread\_create()** :- create a new thread

Description : The **pthread\_create()** function starts a new thread in the calling process. The new thread starts execution by invoking **start\_routine()**; **arg** is passed as the sole argument of **start\_routine()**.

2. **pthread\_exit()** :-terminate calling thread

Description : The **pthread\_exit()** function terminates the calling thread and returns a value via **retval** that (if the thread is joinable) is available to another thread in the same process that calls **pthread\_join**.

3. **pthread\_join()** :-join with a terminated thread

Description : The **pthread\_join()** function waits for the thread specified by **thread** to terminate. If that thread has already terminated, then **pthread\_join()** returns immediately. The thread specified by **thread** must be joinable

4. **pthread\_self()** :-obtain ID of the calling thread

Description : The **pthread\_self()** function returns the ID of the calling thread. This is the same value that is returned in **\*thread** in the **pthread\_create(3)** call that created this thread.

5. **pthread\_cancel()** :-send a cancellation request to a thread

Description : The **pthread\_cancel()** function sends a cancellation request to the thread **thread**. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.

6. **pthread\_detach()** :-detach a thread

Description : The **pthread\_detach()** function marks the thread identified by **thread** as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

7. **pthread\_equal()** :-compare thread IDs

Description : The **pthread\_equal()** function compares two thread identifiers.

8. **pthread\_mutex\_init()** :- initialise or destroy a mutex

Description : The **pthread\_mutex\_init()** function initialises the mutex referenced by **mutex** with attributes specified by **attr**. If **attr** is NULL, the default mutex attributes are

used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

9. `pthread_mutex_destroy()` :- destroy and initialize a mutex

Description : The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

10. `pthread_mutex_lock()` :- Lock a mutex

Description : The `pthread_mutex_lock()` function locks the mutex object referenced by `mutex`. If the mutex is already locked, then the calling thread blocks until it has acquired the mutex. When the function returns, the mutex object is locked and owned by the calling thread.

11. `pthread_mutex_trylock()` :- Attempt to lock a mutex

Description : The `pthread_mutex_trylock()` function attempts to lock the mutex `mutex`, but doesn't block the calling thread if the mutex is already locked.

12. `pthread_mutex_unlock()` :- Unlock a mutex

Description : The `pthread_mutex_unlock()` function unlocks the mutex `mutex`. The mutex should be owned by the calling thread. If there are threads blocked on the mutex then the highest priority waiting thread is unblocked and becomes the next owner of the mutex.

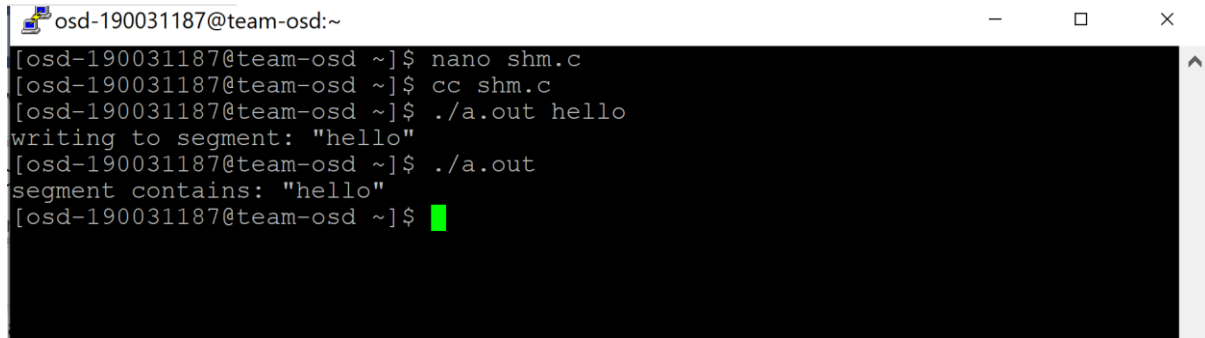
**In-Lab****1. System V shared memory.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024 /* make it a 1K shared memory segment */
int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;
    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("shm.c", 'R')) == -1)
    {
        perror("ftok");
        exit(1);
    }
    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
    {
        perror("shmget");
        exit(1);
    }
    /* attach to the segment to get a pointer to it:*/
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1))
    {
        perror("shmat");
        exit(1);
    }
    /* read or modify the segment, based on the command line: */
    if (argc == 2)
    {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
    {
        printf("segment contains: \"%s\"\n", data);
    }
    /* detach from the segment: */
    shmctl(shmid, IPC_RMID, NULL);
}
```

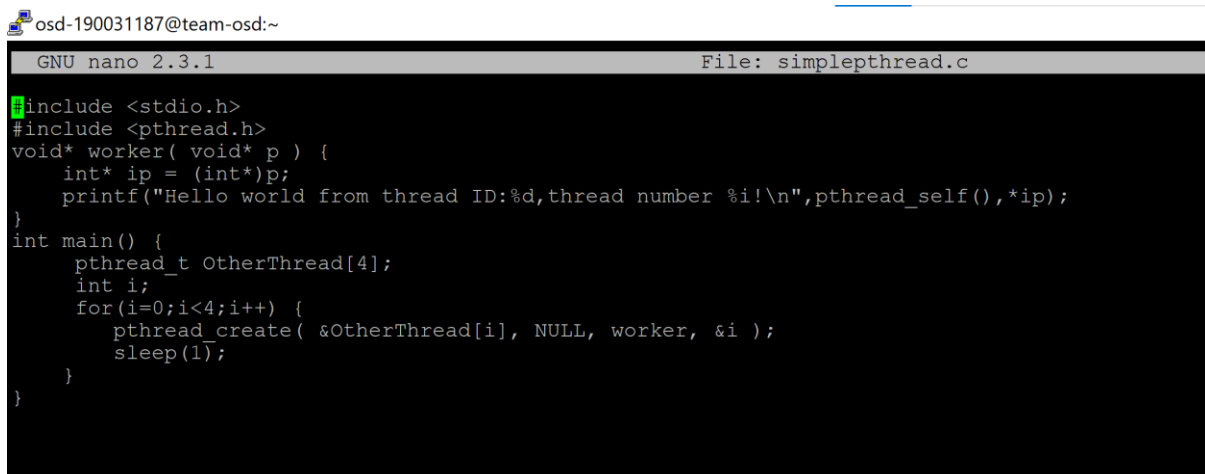
```
if (shmdt(data) == -1) {  
    perror("shmdt");  
    exit(1);  
}  
}
```

### OUTPUT



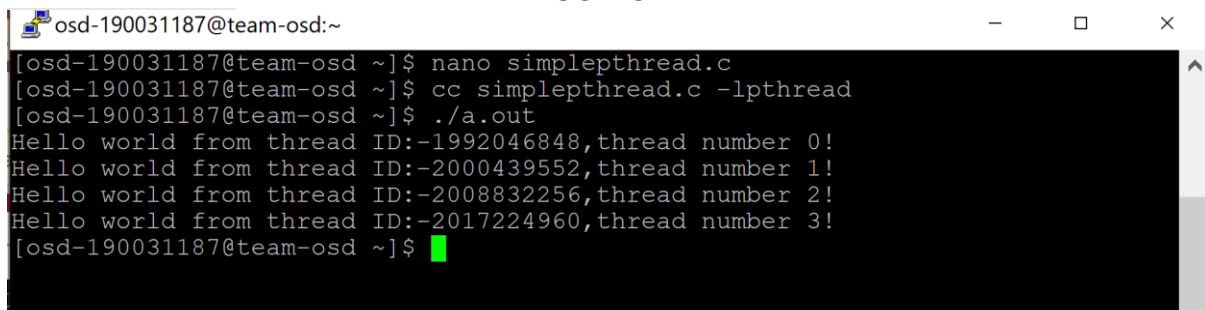
```
osd-190031187@team-osd:~  
[osd-190031187@team-osd ~]$ nano shm.c  
[osd-190031187@team-osd ~]$ cc shm.c  
[osd-190031187@team-osd ~]$ ./a.out hello  
writing to segment: "hello"  
[osd-190031187@team-osd ~]$ ./a.out  
segment contains: "hello"  
[osd-190031187@team-osd ~]$
```

2. Write a program to create 5 pthreads and display Hello world. Main thread should wait until all new threads are terminated. Use Simpler Argument Passing to a Thread.



```
GNU nano 2.3.1 File: simplepthread.c  
#include <stdio.h>  
#include <pthread.h>  
void* worker( void* p ) {  
    int* ip = (int*)p;  
    printf("Hello world from thread ID:%d,thread number %i!\n",pthread_self(),*ip);  
}  
int main() {  
    pthread_t OtherThread[4];  
    int i;  
    for(i=0;i<4;i++) {  
        pthread_create( &OtherThread[i], NULL, worker, &i );  
        sleep(1);  
    }  
}
```

### OUTPUT



```
osd-190031187@team-osd:~  
[osd-190031187@team-osd ~]$ nano simplepthread.c  
[osd-190031187@team-osd ~]$ cc simplepthread.c -lpthread  
[osd-190031187@team-osd ~]$ ./a.out  
Hello world from thread ID:-1992046848,thread number 0!  
Hello world from thread ID:-2000439552,thread number 1!  
Hello world from thread ID:-2008832256,thread number 2!  
Hello world from thread ID:-2017224960,thread number 3!  
[osd-190031187@team-osd ~]$
```

## Post-Lab

### 1. System V message queues.

Message queues:

Message queues are one of the interprocess communication mechanisms available under Linux. Message queues, shared memory and semaphores are normally listed as the three interprocess communication mechanisms under Linux. Semaphores, though, are really for process synchronization. In practice, shared memory, aided by semaphores, makes an interprocess communication mechanism. Message queues is the other interprocess communication mechanism.

System V and POSIX Message queues:

There are two varieties of message queues, System V message queues and POSIX message queues. Both provide almost the same functionality but system calls for the two are different. System V message queues have been around for a long time, since the UNIX systems of 1980s and are a mandatory requirement of Unix-certified systems. POSIX message queues (and the complete POSIX IPC calls) were introduced in 1993 and are still an optional requirement of Unix-certified systems. This tutorial is for System V message queues.